```
from enum import Enum
import re


class TokenType(Enum):
    WHITESPACE = "(?<=[^\\s\\\\]) (?=[^\\s\\\\])"
    NEWLINE = "[\n]+"
    TAB = "[\t]+|(\\\\G|^) {4}"
    LEFTPAREN = "\\("
    RIGHTPAREN = "\\)"
    COLON = "\\:"
    EQUAL = "(?<!=)=(?!=)"
    BINARYOP = "[*|/|+|-]"
    NUMBER = "[0-9]+"
    KEYWORD = "public|private|static|return|new"
    IDENTIFIER = "(\\w+)"


class Token:
    def __init__(self, token_type, value):
        self.token_type = token_type
        self.value = value

    def __repr__(self):
        return self.token_type.name + " (" + self.value + ")"


class Lexer:
    def __init__(self, data):
        self.data = data
        self.tokens = list()

    def lex(self):
        pos = 0
        while pos < len(self.data):
            for pattern, t in [[i.value, str(i)] for i in TokenType]:
                regex = re.compile(pattern)
                match = regex.match(self.data, pos)
                if match:
                    value = match.group(0)
                    if t:
                        self.tokens.append(Token(TokenType(pattern), value))
                    else:
                        print(value)
                        print(t)
                    pos = match.end(0)
                    break
            else:
                raise SyntaxError("Token not found: " + self.data[pos:])
        return self.tokens

```

```
from enum import Enum
from rlexer import TokenType
from rlexer import Token


class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.main_node = None
        self.lines = []
        self.nodes = []

    def parse(self):
        self.lines = self.make_lines()
        for line in self.lines:
            expression = line.to_expressions()
            self.nodes.append(expression.to_nodes())
        self.merge_nodes()
        self.main_node.print()
        return self.main_node

    def merge_nodes(self):
        self.main_node = self.nodes[0]
        for node in self.nodes[1:]:
            tabs = node.tabs
            parent = self.main_node
            for i in range(tabs - 1):
                parent = parent.children[-1]
            parent.add_child(node)
            node.set_parent(parent)
        self.main_node.set_level([])

    def make_lines(self):
        lines = []
        line = []
        for token in self.tokens:
            if token.token_type == TokenType.NEWLINE:
                lines.append(Line(line))
                line = []
            elif token.token_type != TokenType.WHITESPACE:
                line.append(token)
        return lines


class Node:
    def __init__(self, token, node_type):
        self.value = token.value
        self.token = token
        self.node_type = node_type
        self.children = []
        self.parent = None
        self.level = []
        self.tabs = None

    def add_child(self, child):
        self.children.append(child)

    def set_parent(self, parent):
        self.parent = parent

    def set_level(self, base, i=0):
        self.level += base
        self.level.append(i)
        for i, child in enumerate(self.children):
            child.set_level(self.level, i)
```

```
66
67      def print(self):
68          print(self.level, self.node_type.name, self.token.value)
69          if self.children:
70              for c in self.children:
71                  c.print()
72
73
74  class NodeType(Enum):
75      ASSIGN = 0
76      CALL = 1
77      BIN = 2
78      DECLARE = 3
79      NUMBER = 4
80      VARIABLE = 5
81
82
83  class Expression:
84      def __init__(self):
85          self.contents = []
86          self.tabs = 0
87
88      def __len__(self):
89          return len(self.contents)
90
91      def append(self, i):
92          self.contents.append(i)
93
94      def get_contents(self):
95          return self.contents
96
97      def to_nodes(self):
98          child_nodes = []
99          node = None
100         for element in self.contents:
101             node_type = None
102             if type(element) == Token:
103                 if len(self) == 2 and element.token_type == TokenType.IDENTIFIER and type(
104                         self.contents[1]) == Expression:
105                     node_type = NodeType.CALL
106                 elif element.token_type == TokenType.EQUAL:
107                     node_type = NodeType.ASSIGN
108                 elif element.token_type == TokenType.BINARYOP:
109                     node_type = NodeType.BIN
110                 elif element.token_type == TokenType.KEYWORD:
111                     node_type = NodeType.DECLARE
112                 elif element.token_type == TokenType.NUMBER:
113                     node_type = NodeType.NUMBER
114                 elif element.token_type == TokenType.IDENTIFIER:
115                     node_type = NodeType.VARIABLE
116                 tmp_node = Node(element, node_type)
117                 if node_type == NodeType.NUMBER or node_type == NodeType.VARIABLE:
118                     child_nodes.append(tmp_node)
119                 else:
120                     node = tmp_node
121             elif type(element) == Expression:
122                 tmp_node = element.to_nodes()
123                 child_nodes.append(tmp_node)
124         if node is None:
125             if child_nodes:
126                 node = child_nodes[0]
127                 child_nodes = []
128             else:
129                 return None
130         node.tabs = self.tabs
```

```
131         for n in child_nodes:
132             node.add_child(n)
133             n.set_parent(node)
134         return node
135 
136 
137 class Line:
138     def __init__(self, line):
139         self.line = line
140         self.index = 0
141 
142     def has_next(self):
143         return self.index < len(self.line)
144 
145     def next(self):
146         self.index += 1
147         return self.line[self.index - 1]
148 
149     def to_expressions(self):
150         expression = Expression()
151         while self.has_next():
152             element = self.next()
153             if element.token_type == TokenType.TAB:
154                 expression.tabs += 1
155             elif element.token_type == TokenType.KEYWORD:
156                 expression.append(element)
157             elif element.token_type == TokenType.IDENTIFIER:
158                 expression.append(element)
159             elif element.token_type == TokenType.NUMBER:
160                 expression.append(element)
161             elif element.token_type == TokenType.BINARYOP or element.token_type ==
    TokenType.EQUAL or element.token_type == TokenType.LEFTPAREN:
162                 if element.token_type != TokenType.LEFTPAREN:
163                     expression.append(element)
164                 next_expression = self.to_expressions()
165                 if len(next_expression) == 1 and element.token_type != TokenType.LEFTPAREN
    :
166                     expression.append(next_expression.get_contents()[0])
167                 elif len(next_expression) != 0:
168                     expression.append(next_expression)
169             elif element.token_type == TokenType.RIGHTPAREN:
170                 return expression
171         return expression
172 
```